Consensus-Driven Distributable Thread Scheduling in Networked Embedded Systems

Jonathan S. Anderson[1], Binoy Ravindran[1], and E. Douglas Jensen[2]

[1] Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg Virginia, 24061, USA
{andersoj, binoy}@vt.edu

[2] The MITRE Corporation
Bedford, Massachusetts, 01730, USA
jensen@mitre.org

Abstract

We demonstrate a consensus utility accrual scheduling algorithm for distributable threads with run-time uncertainties in execution time, arrival models, and node crash failures. The DUA-CLA algorithm's message complexity ($O(fn)$), lower time complexity bound ($O(D+fd+nk)$), and failure-free execution time ($O(D+nk)$) are established, where D is the worst-case communication delay, d is the failure detection bound, n is the number of nodes, and f is the number of failures. The "lazy-abort" property is shown — abortion of currently-infeasible tasks is deferred until timely task completion is impossible. DUA-CLA also exhibits "schedule-safety" — threads proposed as feasible for execution by a node which fails during the decision process will not cause an otherwise-feasible thread to be excluded. These properties mark improvements over earlier strategies in common- and worst-case performance. Quantitative results obtained from our Distributed Real-Time Java implementation validate properties of the algorithm.

1 Introduction

1.1 Dynamic Distributed Real-Time Systems

Distributed real-time systems such as those found in industrial automation, net-centric warfare (NCW), and military surveillance must support for timely, end-to-end activities. Timeliness includes application-specific acceptability of end-to-end time constraint satisfaction, and of the predictability of that satisfaction. These activities may include computational, sensor, and actuator steps which levy a causal ordering of operations, contingent on interactions with physical systems. Such end-to-end tasks may be represented in a concrete distributed system as: chains of (a) nested remote method invocations; (b) publish, receive steps in a publish-subscribe framework; (c) event occurrence and event handlers.

Dynamic Systems. The class of distributed real-time systems under consideration here, typified by NCW applications [1], is characterized by dynamically uncertain execution properties due to transient and persistent local overloads,

uncertain task arrival patterns and rates, uncertain communication delays, node failures, and variable resource demands. However, while local activities in these systems may have timeliness requirements with sub-second magnitudes, end-to-end tasks commonly have considerably larger magnitudes of milliseconds to multiple minutes. Despite larger timeliness magnitudes, these activities are mission-critical and require the strongest assurances possible under the circumstances.

End-to-End Context. In order to make resource allocation decisions in keeping with end-to-end activity requirements, some representation of these parameters and the current state of the end-to-end activity must be provided. The *distributable thread* abstraction provides such an extensible model for reasoning about end-to-end activity behavior. Distributable threads (hereafter, simply *threads*) appeared first in the Alpha OS [2] and were adopted in Mach 3.0 [3] and Mk7.3 [4]. Recently, this abstraction has served as the basis for RT-CORBA 2.0 [5] and Sun's emerging Distributed Real-Time Specification for Java (DRTSJ) [6], where threads form the primary programming abstraction for concurrent, distributed activities.

Time Constraints for Overloaded Systems. In underloaded systems it is sufficient to provide an assessment of the *urgency* of an activity, typically in the form of a *deadline*. For these scenarios, known-optimal algorithms (e.g., EDF [7]) exist to meet all deadlines (given some restrictions.) When a system is overloaded, resource managers must decide *which* subset of activities to complete, and with *what degree of timeliness*.

This requires the system to be aware of the relative *importances* activities. We consider the *time/utility function* (TUF) timeliness model [8], in which the utility of completing an activity is given as a function of its completion time. This paper is confined to downward step-shaped TUFs, wherein an activity's utility is a constant U_i when the task is completed before a deadline t; no utility is gained for tasks after their deadline.

Utility Accrual Scheduling. When time constraints are expressed as TUFs, resource allocation decisions may be expressed in terms of *utility accrual* (UA) criteria. A common UA criteria is *maximize summed utility*, in which resource allocation decisions are made such that the total summed utility accrued is maximized. Several such UA scheduling and sequencing heuristics have been investigated (e.g., [9, 10]). Such algorithms for activities described by downward-step TUFs equate to EDF during underload conditions, achieving optimum schedules. During overloads, these algorithms favor higher-utility activities over those with lower-utility. Resulting "best-effort" adaptive behavior exhibits graceful degradation as load increases, shedding low utility work irrespective of its urgency.

1.2 Contributions and Related Work

The central contributions of this paper are: (*a*) the Distributed Utility Accrual - Consensus-based Lazy Abort (DUA-CLA) scheduling algorithm; (*b*) bounds on DUA-CLA's timeliness, message efficiency, and optimality behavior in a variety of conditions; (*c*) implementation of DUA-CLA in the DRTSJ middleware; and (*d*) experimental results illustrating the validity of the theoretical results.

This paper presents significant progress on work published by the authors in [11], expanding the theoretical performance envelope in two ways: First, the "lazy-abort" property is introduced (Theorem 6), relaxing conservative task-abortion behavior present in earlier work, while maintaining asymptotic execution times and performance assurances. Second, DUA-CLA is shown (see Theorem 7) to be "schedule-safe" in the presence of failures during distributed rescheduling. This property mitigates pessimism in feasibility assessments due to failures which results in unnecessary task rejection during partial failure.

The DUA-CLA algorithm represents a unique approach to distributable thread scheduling in two respects. First, it unifies scheduling with a fault-tolerance strategy. Previous work on distributable thread scheduling [12,13] addresses only the scheduling problem, with fault tolerance dealt with by separate *thread integrity protocols* [12–14]. While this provides admirable separation of concerns, scheduling and integrity operations become tightly intertwined in distributed systems where failures are prevalent.

Second, DUA-CLA takes a *collaborative* approach to the scheduling problem, rather than requiring nodes independently to schedule tasks without knowledge of other nodes' states. Global scheduling approaches wherein a single, centralized scheduler makes all scheduling decisions have been proposed and implemented. DUA-CLA takes a *via media*, improving independent node scheduler decision-making with partial knowledge of global system state.

Little work has been done on collaborative distributed scheduling for real-time systems. The RT-CORBA 2.0 specification [5] envisions such an approach, enumerating it as the third of its four "cases" for distributed scheduling. Poledna, et. al. consider a consensus-based sequencing approach for operations on replicas to ensure consistency [15]. In a similar vein, Gammar and Kammoun present a consensus algorithm for ensuring database properties such as serializability and consistency in real-time transactional systems [16]. None of these directly address the question of end-to-end causal activity scheduling.

2 The DUA-CLA Algorithm

2.1 Models

Distributable Thread Abstraction. Threads execute in local and remote objects by location-independent invocations and returns. A thread begins its execution by invoking an object operation. The object and the operation are specified when the thread is created. The portion of a thread executing an object operation is called a *thread segment*. Thus, a thread can be viewed as being composed of a concatenation of thread segments.

A thread's initial segment is called its *root* and its most recent segment is called its *head*. The head of a thread is the only segment that is active. A thread can also be viewed as a sequence of *sections*, where a section consists of all contiguous thread segments on a node. Further details of the thread model can be found in [5,6,13]. Execution time estimates (possibly inaccurate) of the sections

of a thread areknown when the thread arrives.The application is thus comprised of a set of threads, denoted $\mathbf{T} = \{T_1, T_2, T_3, \ldots\}$, with sections $[S_1^i, S_2^i, \ldots, S_k^i]$.
Timeliness Model. We specify the time constraint of each thread using a TUF. A thread T_i's unit downward step TUF is denoted as $U_i(t)$, which has a initial time I_i, which is the earliest time for which the TUF is defined, a termination time X_i, which, for a downward step TUF, is its discontinuity point, and a constant utility U_t. $U_i(t) > 0, \forall t \in [I_i, X_i]$ and $U_i(t) = 0, \forall t \notin [I_i, X_i], \forall i$.
System and Failure Models Our system and failure models follow that of [17]. We consider a system model where a set of processing *nodes* are denoted by the totally-ordered set $\Pi = \{1, 2, \ldots, n\}$. We consider a single hop network model (e.g., a LAN), with nodes interconnected through a hub or a switch. The system is assumed to be (partially) synchronous in that there exists an upper bound D on the message delivery latency. A reliable message transmission protocol is assumed; thus messages are not lost or duplicated. Node clocks are assumed to be perfectly synchronized, for simplicity in presentation, though DUA-CLA can be extended to clocks that are nearly synchronized with bounded drift rates. As many as f_{max} nodes may crash arbitrarily. The actual number of node crashes is denoted as $f \leq f_{max}$. Nodes that do not crash are called *correct*.

Each node is assumed to be equipped with a perfect failure detector [18] that provides a list of nodes deemed to have crashed. If a node q belongs to such a list of node p, then node p is said to *suspect* node q. The failure detection time [19] $d \leq D$ is bounded. Similar to [17], for simplicity in presentation, we assume that D is a multiple of d. Failure detectors are assumed to be (a) *accurate*—i.e., a node suspects node q only if q has previously crashed; and (b) *timely*—i.e., if node q crashes at time t, then correct nodes permanently suspect q within $t + d$.

2.2 Rationale and Design

Our primary scheduling objective is to maximize total utility accrued by all the threads Further, the algorithm must provide assurances on the satisfaction of thread termination times in the presence of (up to f_{max}) crash failures. Moreover, the algorithm must exhibit the UA best-effort property described in Section 1.

Definition 1 (Current and Future Head Nodes). *The current head node of a thread T_i is the node where T_i is currently executing (i.e., where T_i's head is currently located). The future head nodes of a thread T_i are those nodes where T_i will make remote invocations in the future.*

The crash of a node p affects other nodes in the system in three possible ways: (a) p may be the current head node of one or more threads; (b) p may be the future head node of one or more threads; and (c) p may be the current and future head node of one or more threads.

If p is only the current head node of one or more threads, then all its future head nodes are immediately affected when p crashes, since they can now release allocated processor time. This implies that when a node p crashes, a system-wide decision must be made regarding which subset of threads are eligible for

execution in the system—referred to as an *execution-eligible thread set*. This decision must be made in the presence of failures since nodes may crash while that decision is being made. We formulate this problem as a *consensus* problem [20] with the following properties: (a) If a correct node decides an eligible thread set $\mathcal{T}$, then some correct node proposed $\mathcal{T}$;[3] (b) Nodes (correct or not) do not decide different execution-eligible sets (*uniform agreement*); (c) Every correct node eventually decides (i.e., termination).

How can a node propose a set of threads which are eligible for execution? The task model is dynamic and future scheduling events cannot be considered at a scheduling event.[4] Thus, the execution-eligible thread set must be constructed exploiting the current system knowledge. Since the primary scheduling objective is to maximize the total thread accrued utility, a reasonable heuristic for determining the execution-eligible thread set is a "greedy" strategy: Favor "high return" threads over low return ones, and complete as many of them as possible before thread termination times.

The potential utility that can be accrued by executing a thread section on a node defines a measure of that section's "return on investment." We measure this using a metric called the *Potential Utility Density* (or PUD). On a node, a thread section's PUD measures the utility that can be accrued per unit time by immediately executing the section on the node.

Thus, each node iteratively examines thread sections in its local ready queue for potential inclusion in a feasible (local) schedule in order of decreasing section PUDs. For each section, the algorithm examines whether that section can be completed early enough, allowing successive sections of the thread to also be completed early enough, to allow the entire thread to meet its termination time. We call this property the *feasibility* of a section. Infeasible sections are not included in the working schedule. This approach requires a decomposition of the thread's deadline, which is computed at arrival time using the following conservative approach: The section termination times of a thread T_i with k sections are given by:

$$S_j^i.tt = \begin{cases} T_i.tt & j = k \\ S_{j+1}^i.tt - S_{j+1}^i.ex - D & 1 \leq j \leq k-1 \end{cases} \quad (1)$$

where $S_j^i.tt$ denotes section S_j^i's termination time, $T_i.tt$ denotes T_i's termination time, and $S_j^i.ex$ denotes the estimated execution time of section S_j^i.

Thus, the local schedule constructed by a node p is an ordered list of a subset of sections in p's ready queue that can be feasibly completed, and will likely result in high local accrued utility (due to the greedy nature of the PUD heuristic). The set of threads, say T_p, of these sections included in p's schedule is proposed by p as those that are eligible for system-wide execution, from p's standpoint. However, not all threads in T_p may be eligible for system-wide execution, because

[3] This property is stronger than the conventional *Uniform Validity* property, and therefore requires additional constraints.

[4] A "scheduling event" is any event that invokes the scheduling algorithm.

the current and/or future head nodes of some of those threads may crash. Consequently, the set of threads that are eligible for system-wide execution is that subset of threads with no absent sections from their respective current and/or future head node schedules.

2.3 Algorithm Description

The DUA-CLA algorithm that we present is derived from Aguilera *et. al*'s time-optimal, early-deciding, uniform consensus algorithm [17]. A pseudo-code description of DUA-CLA on each node i is shown in Algorithm 1.

Algorithm 1: DUA-CLA: Code for each node i

```
1  input: σ_r^i; output: σ_i; // σ_r^i: unordered ready queue of node i's sections; σ_i:
       schedule
2  Initialization: Σ_i = ∅; ω_i = ∅; max_i = 0;
3  σ_i = ConstructLocalSchedule(σ_r^i);
4  send(σ_i, i) to all;
5  upon receive (σ_j, j) until 2D do // After time 2D, consensus begins
6    Σ_i = Σ_i ∪ σ_j;
7  ω_i = DetermineSystemWideFeasibleThreadSet(Σ_i);
8  upon receive (ω_j, j) do
9    if j > max_i then max_i = j; ω_i = ω_j;
10 at time (i − 1)d do
11   if suspect j for any j : 1 ≤ j ≤ i − 1 then
12     ω_i = UpdateFeasibleThreadSet(Σ_i);
13     send(ω_i, i) to all;
14 at time (j − 1)d + D for every j : 1 ≤ j ≤ n do
15   if trust j then decide ω_i;
16 UpdateSectionSet(ω_i, σ_r^i);
17 σ_i = ConstructLocalSchedule(σ_r^i);
18 return σ_i;
```

The algorithm is invoked at a node i at the scheduling events including 1) creation of a thread at node i and 2) inclusion of a node k into node i's suspect list by i's failure detector.

When invoked, a node i first constructs a local schedule (`ConstructLocalSchedule()`), sending this schedule (σ_i, i) to all nodes. Recipients respond immediately by constructing local section schedules and sending them to all nodes. When node i receives a schedule (σ_j, j), it includes that schedule into a schedule set Σ_i. Thus, after $2D$ time units, all nodes have a schedule set containing all schedules received.

A node i then determines its consensus decision, computed from Σ_i as the subset of threads with no sections absent from node schedules in Σ_i. Node i uses a variable ω_i to maintain its consensus decision.

The algorithm divides time in rounds of duration d, where the ith round corresponds to the time interval $[(i-1)d, id)$. At the beginning of round i, node i checks whether it suspects *any* of the nodes with smaller node ID. If so, it computes a new ω_i using `UpdateFeasibleThreadSet()` (see Algorithm 2),

sending (ω_i, i) to all nodes. Note that the messages sent in a round could be received in a higher round since $D > d$.

Algorithm 2: `UpdateFeasibleThreadSet`

```
1: input: ω_i; output: ω'_i;      // ω'_i: feasible section set with sections on failed
       nodes removed
2: initialize:  ω'_i = ω_i
3: for each section S^t_j ∈ ω_i do
4:     if  suspect j then ω'_i = ω'_i \ S^t_j;
5:     return σ'_i;
```

Each node i maintains a variable max_i that contains the largest node ID from which it has received a consensus proposal. When a node i receives a proposed execution-eligible thread set (ω_j, j) that is sent from another node j with an ID that is larger than max_i (i.e., $j > max_i$), node i updates its consensus decision to thread set ω_j and max_i to j. At times $(j-1)d + D$ for $j = 1, \ldots, n$, node i is guaranteed to have received potential consensus proposals from node j. At these times, i checks whether j has crashed; if not, i arrives at its consensus decision on the thread set ω_i.

Node i then updates its ready queue σ_r^i by removing those sections whose threads are absent in the consensus decision ω_i. The updated ready queue is used to construct a new local schedule σ_i, the head section of which is subsequently dispatched for execution.

2.4 Constructing Section Schedules

We now describe the algorithm `ConstructLocalSchedule()` and its auxiliary functions. Since this algorithm is not a distributed algorithm *per se*, we drop the suffix i from notations σ_r^i (input unordered list) and σ_i (output schedule), and refer to them as σ_r and σ, respectively. Sections are referred to as S_i, for $i = 1, 2, \ldots$

Algorithm 3 describes the local section scheduling algorithm. When invoked at time t_{cur}, the algorithm first checks the feasibility of the sections. First, if the earliest conceivable execution (the current time) of segment will still miss the termination time, the algorithm aborts the section. If the earliest predicted completion time of a section is later than its termination time, it is removed from this round's consideration. The sections considered for insertion into σ in order of decreasing PUD, which is maintained in order of non-decreasing section termination times. After inserting a section S_i, the schedule σ is tested for feasibility.[5] If σ becomes infeasible, S_i is removed. After examining all sections, the ordered list σ is returned.

Algorithm 3 includes those sections likely to result in high total utility (due to the greedy nature of the PUD heuristic). Further, since the invariant of schedule feasibility is preserved throughout the examination of sections, the output

[5] A schedule σ is feasible if the predicted completion time of each section $S_i \in \sigma$ does not exceed S_i's termination time. For explicit pseudo-code for a linear-time implementation, see Algorithm 3 in [11].

Algorithm 3: `ConstructLocalSchedule()`

```
1:  input: σ_r; output: σ;
2:  Initialization: t := t_cur, σ := ∅;
3:  for each section S_i ∈ σ_r do
4:      if current_time + S_i.ex > S_i.tt then
            abort(S_i)
5:      if S^i_{j-1}.tt + D + S^i_j.ex > S^i_j.tt then
6:          σ_r = σ_r \ S_i;
        else
7:          S_i.PUD = U_i (t + S_i.ex) / S_i.ex;
8:  σ_tmp := sortByPUD(σ_r);
9:  for each section S_i ∈ σ_tmp from head to tail do
10:     if S_i.PUD > 0 then
11:         Insert(S_i, σ, S_i.tt);
12:         if Feasible(σ)=false then
13:             Remove(S_i, σ, S_i.tt);
14:     else break;
15: return σ;
```

schedule is always a feasible schedule. During underloads, schedule σ will always be feasible in (Algorithm 3), the algorithm will never reject a section, and will produce a schedule which is the same as that produced by EDF (where deadlines are equal to section termination times). This schedule will meet all section termination times during underloads. During overloads, one or more low-PUD sections will not be included. These rejected sections are less likely to contribute a total utility larger than that contributed by accepted sections. The asymptotic complexity of Algorithm 3 is dominated by the nested loop with calls `Feasible()`, resulting in a cost of $O(k^2)$.

3 Algorithm Properties

We now establish DUA-CLA's timeliness and execution time properties in both absence and presence of failures. We first describe DUA-CLA's timeliness property under crash-free runs. The proof of this and some future results are elided for space, but may be found in the full version of the paper.[6] [17]

Theorem 1 *If all nodes are underloaded and no nodes crash (i.e., $f_{max} = 0$), DUA-CLA meets all thread termination times, yielding optimum total utility.*

Theorem 2 *DUA-CLA achieves (uniform) consensus (i.e., uniform validity, uniform agreement, termination) on the system-wide execution-eligible thread set in the presence of up to f_{max} failures.*

Theorem 3 *DUA-CLA's time complexity is $O(D + fd + nk)$ and message complexity is $O(fn)$.*

[6] Full paper available at: `http://www.real-time.ece.vt.edu/euc07.pdf`

Theorem 4 *If $n-f$ nodes (i.e., correct nodes) are underloaded, then DUA-CLA meets the termination times of all threads in its execution-eligible thread set.*

To establish the algorithm's best-effort property (Section 1), we define NBI:

Definition 2. *Consider a distributable thread scheduling algorithm $\mathcal{A}$. Let a thread T_i be created at a node at a time t with the following properties: (a) T_i and all threads in $\mathcal{A}$'s execution-eligible thread set at time t are not feasible (system-wide) at t, but T_i is feasible just by itself; and (b) T_i has the highest PUD among all threads in $\mathcal{A}$'s execution-eligible thread set at time t. Now, $\mathcal{A}$'s non-best-effort time interval, denoted $NBI_{\mathcal{A}}$, is defined as the duration of time that T_i will have to wait after t, before it is included in $\mathcal{A}$'s execution-eligible thread set. Thus, T_i is assumed to be feasible at $t + NBI_{\mathcal{A}}$.*

We now describe the NBI of DUA-CLA and other distributable thread scheduling UA algorithms including DASA [9], LBESA [10], and AUA [12] under crash-free runs.

Theorem 5 *Under crash-free runs (i.e., $f_{max} = 0$), the worst-case NBI of DUA-CLA is $3D + \delta$, DASA's and LBESA's is δ, and that of AUA is $+\infty$.*

In order to further characterize the algorithm's best-effort behavior in the presence of failures, we introduce definitions for *Lazy-Abort* behavior and *Schedule Safety*:

Definition 3. *A collaborative distributable thread scheduling algorithm is said to Lazy-Abort if it delays abortion of a segment until it would be infeasible if it were the only thread in the system.*

Theorem 6 *DUA-CLA demonstrates Lazy-Abort behavior. Sections are only aborted in `ConstructLocalSchedule()` (Algorithm 3), and then only when the segment would exceed its deadline if it were executed immediately. If this is the case, the Lazy-Abort condition is met. Consequently, transient perceived overloads which resolve through node failures or pessimistic execution time evaluations do not cause overly-aggressive abortion of future threads.*

Definition 4. *A consensus-based distributable thread scheduling algorithm is said to exhibit Schedule Safety if it never allows the presence of a remote segment S_f in the global feasible set to render infeasible a local segment S_l if the node hosting S_f is known to have failed during consensus.*

Theorem 7 *DUA-CLA demonstrates schedule-safety despite failures during the distributed scheduling event. The algorithm evaluates feasibility of local segments on node i based on the section set updated in the call to `UpdateSectionSet()` in Algorithm 1. If any nodes j with $1 < j < i$ is suspected by i, then i removes all segments on j from ω_i. (Furthermore, at time $D + fd$, all nodes will receive this reduced proposed section set.) Therefore, no locally feasible thread segments will be rendered infeasible by erroneous inclusion of segments from j.*

4 Implementation Experience

A major objective this work was to bridge the gap between theoretical consideration and the practicalities of implementation. In particular, we constructed experiments to uncover time complexity constants implicit in Theorem 3. Single-node task sets are compared to distributed sets in the presence of underloads as well as overloads, in failure-free as well as the $f = f_{max}$ case. We explore algorithm overhead by comparing well-known single-node scheduling disciplines to a degenerate case of our collaborative approach.

DUA-CLA was implemented on the DRTSJ reference implementation (DRTSJ-RI), consisting of Apogee's Aphelion-based DRTSJ-compliant JVM, executing on Pentium-class machines with Ubuntu Edgy Linux (kernel 2.6.17 with real-time extensions). Nodes were connected via 10Mbps Ethernet through a Linux-based dynamic switch, configured with the `netem` network emulation module [21] to introduce controlled communication delay. Failure detector traffic, experimental control traffic, and normal communication traffic were allocated to priority bands configured to simulate communication delays consonant with the system model described in Section 2.1, resulting in particular in the relationship $D \gg d$ between common communication latency and failure detection latency.

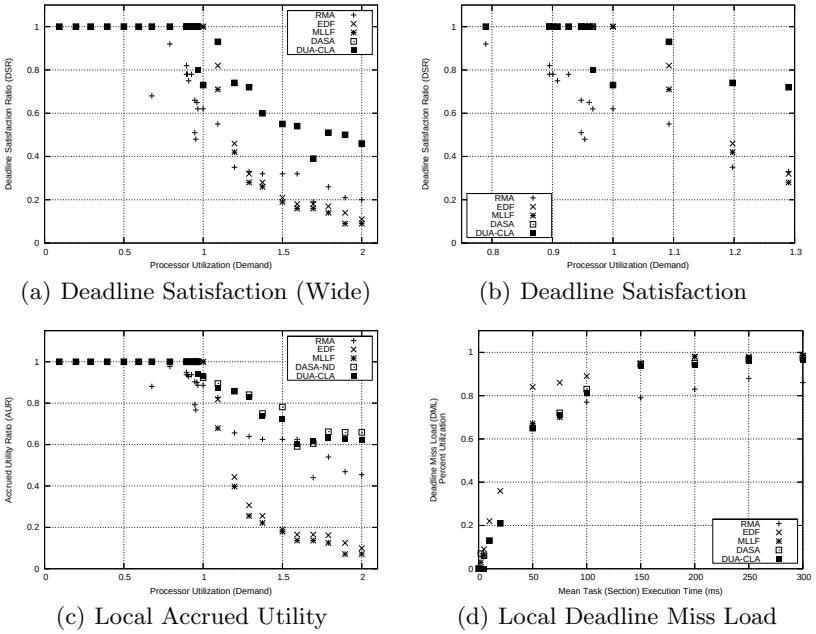
A heartbeat fast-failure detector was implemented as a small, pure RTSJ application [22] with the highest execution eligibility on the node, and not preemptible by the garbage collector. A heartbeat period of 1ms and evaluation period of 3ms were chosen to match the magnitudes of task execution times. Extensive measurements of latency d and application message delay D were made across a range of CPU and network utilization, with no failure detection latency greater than 2.98 milliseconds; therefore we use $d = 2.98$ms for the following DUA-CLA experiments. Similarly, we measured a worst-case message delay $D \approx 69.87$ms. Both latencies are stable across a range of CPU utilization, closely approximating a perfect fast failure detector.

With this detector, the DUA-CLA algorithm was implemented on top of the DRTSJ Metascheduler, a pluggable scheduling framework enabling user-space scheduling implementations in DRTSJ [6], and previously on QNX [23].

Our experiments took place in the testbed described above, with one DRTSJ-compliant JVM instantiated on each node. Node clocks were synchronized using NTP [24]. The dynamic switch was configured to insert normally-distributed communication delay in application communication traffic. All application communication was via UDP, with reliable messaging provided by the application.

Local Scheduler Performance. In order to establish a baseline for assessing the performance of our scheduler implementation, we compared DUA-CLA to a variety of other scheduling algorithms. In these experiments, each submitted thread consisted of a single segment to be executed on the local node. Since no remote segments appeared in the ready set, no remote scheduling event was triggered and DUA-CLA is functionally equivalent to Clark's DASA algorithm.

Figures 1(a) and 1(b) illustrate deadline satisfaction performance of some UA and non-UA scheduling policies. We use Deadline Satisfaction Ratio (DSR), the ratio of jobs which satisfy their deadline to the total number of jobs released.



(a) Deadline Satisfaction (Wide)

(b) Deadline Satisfaction

(c) Local Accrued Utility

(d) Local Deadline Miss Load

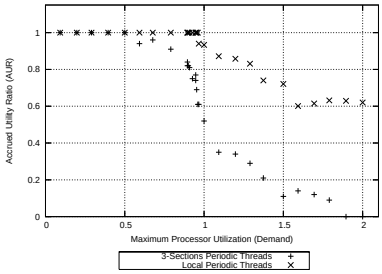
Fig. 1. Local Scheduler Performance: Deadline Satisfaction, AUR, and DML

In the case of this non-distributed experiment, a job is exactly equivalent to a thread segment. A collection of 5 periodic threads were created with relatively-prime periods and random phase offsets. Mean execution time for each job release was varied producing processor demands ranging from 0 to 200%.

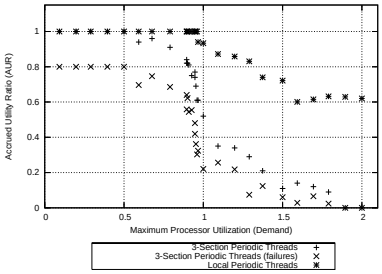
The schedulers presented here include Rate Monotonic Analysis (RMA), Earliest Deadline First, Modified Least-Laxity First (MLLF), Dependent Activity Scheduling Algorithm (DASA), and DUA-CLA. Of these, only DASA and DUA-CLA are utility accrual algorithms. Each algorithm was implemented in the Metascheduler, and each was run with an identical task set for each utilization.

Figure 1(b) provides a detailed look at the "deadline-miss load" region from Figure 1(a), the utilization range at which the scheduling policies begin missing activity deadlines. Theoretically, each of the schedulers shown (with the exception of RMA) should obtain 100% DSR up to 100% load. However, due to middleware overhead activities miss deadlines at lower CPU utilizations. Understanding this overhead as we consider more complex scheduling policies is critical to engineering systems which appropriately trade off scheduling "intelligence" against the additional overhead incurred by more complex policies.

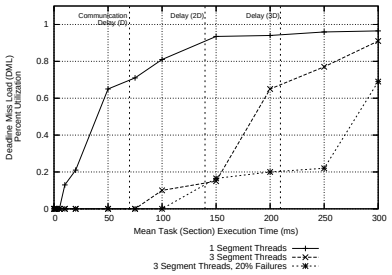
Figure 1(c) captures scheduler performance measured against the UA metric Accrued Utility Ratio (AUR). AUR is the ratio of the accrued utility (sum of the U_i for all completed jobs) to the utility available. Since we have chosen unit-downward step TUFs for these experiments, the AUR and DSR are similar metrics, with AUR appearing as a weighted form of the DSR, with weights U_i.



(a) AUR for 3-Segment Threads



(b) AUR with $f = f_{max} = 0.2n$



(c) Deadline Miss Load for DUA-CLA

Fig. 2. Distributed Scheduler Performance

The reader will note that, while Figures 1(a) and 1(b) indicate that the non-UA policies like RMA sometimes outperform DASA and DUA-CLA in the DSR metric during overloads, Figure 1(c) shows us that this is because RMA is dropping the "wrong" tasks, while the UA policies favor high-utility tasks. It is precisely this behavior we wish to explore in the distributed case, in particular understanding the additional overhead incurred.

Finally, Figure 1(d) characterizes scheduler overhead for each policy by measuring Deadline Miss Load (DML). For each data point, a task execution time (the plot's x-axis) was fixed for every job during a single run. Periods of each periodic task were varied, measuring the resulting utilization and deadline satisfaction. The DML (the y-axis) is that greatest utilization for which the scheduling policy was able to meet all deadlines. The theoretical (zero-overhead) behavior during underload for each policy is a DML of 1.0: these policies should never miss a deadline until the CPU is saturated.

Distributed Scheduler Performance. Our final set of experiments sought to establish the concrete behavior of the DUA-CLA algorithm for qualitative comparison to local scheduling approaches, for validation of the theoretical results above, and to investigate execution timescales for which consensus thread scheduling is appropriate. In each trial, each three-segment periodic thread originates on a common origin node with a short segment, makes an invocation onto one of many server nodes to execute a second segment, then returns to the origin

node to complete in a final segment. We fix the periods, and vary the execution times to produce the utilizations in Figure 2.

In Figure 2(a), we compare the AUR of a collection of one-segment (local) threads to a collection of three-segment threads. As can be seen from the plot, the penalty incurred by collaboration is significant, but the scheduling policy continues to accrue utility through 1.8 fractional utilization. Furthermore, Theorem 1 is borne out by the underloaded portion of Figure 2(a), modulo scheduling overhead. This overhead is explored in detail in the discussion of Figure 2(c).

The behavior of DUA-CLA in the presence of failures is shown in Figure 2(b), wherein we fail f_{max} nodes. Again, the performance of the scheduler suffers, but as shown in Theorem 4, our implementation meets the termination times for all threads remaining on correct nodes.

Finally, we investigate overhead incurred by DUA-CLA across a selection of mean task execution times. Figure 2(c) demonstrates the expected penalty paid in terms of DML for conducting collaborative scheduling. It is clear that the DML for tasks with execution times less than $3D$ suffers because this is the minimal communication delay required to accept a thread's segments for execution.

5 Conclusions and Future Work

The preliminary investigation of consensus-driven collaborative scheduling approach described in this work may be extended in a variety of ways. In particular, algorithmic support for shared resources, deadlock detection and resolution, and quantitative assurances during overload represent worthwhile theoretical questions. Furthermore, improved implementations investigating real-world behavior under failures and with non-trivial abort handling are suggested by the results presented here. An exhaustive look at the practical message complexity would enable broad-based analysis of algorithm design and implementation trade-offs between time complexity and overload schedule quality.

References

1. CCRP: Network centric warfare. `www.dodccrp.org/ncwPages/ncwPage.html`
2. Northcutt, J.D., Clark, R.K.: The Alpha operating system: Programming model. Archons Project Tech. Report 88021, Dept. of Computer Science, Carnegie Mellon, Pittsburgh, PA (Feb 1988)
3. Ford, B., Lepreau, J.: Evolving Mach 3.0 to a migrating thread model. In: USENIX Technical Conference. (1994) 97–114
4. Open Group: MK7.3a Release Notes. Open Group Research Institute, Cambridge, Mass. (Oct 1998)
5. OMG: Real-time CORBA 2.0: Dynamic scheduling. Technical report, Object Management Group (Sept 2001)
6. Anderson, J., Jensen, E.D.: The distributed real-time specification for Java: Status report. In: JTRES. (2006)
7. Chetto, H., Chetto, M.: Some results of the earliest deadline scheduling algorithm. IEEE Transactions on Software Engineering **15**(10) (1989) 466–473

8. Jensen, E.D., et al.: A time-driven scheduling model for real-time systems. In: RTSS. (Dec. 1985) 112–122
9. Clark, R.K.: Scheduling Dependent Real-Time Activities. PhD thesis, CMU (1990)
10. Locke, C.D.: Best-Effort Decision Making for Real-Time Scheduling. PhD thesis, CMU (1986) CMU-CS-86-134.
11. Ravindran, B., Anderson, J.S., Jensen, E.D.: On distributed real-time scheduling in networked embedded systems in the presence of crash failures. In: Proceedings of SEUS 2007. (May 2007)
12. Curley, E., Anderson, J.S., et al.: Recovering from distributable thread failures with assured timeliness in real-time distributed systems. In: SRDS. (2006) 267–276
13. Northcutt, J.D.: Mechanisms for Reliable Distributed Real-Time Operating Systems — The Alpha Kernel. Academic Press (1987)
14. Goldberg, J., Greenberg, I., et al.: Adaptive fault-resistant systems. Technical report, SRI Int'l. (Jan 1995) `http://www.csl.sri.com/papers/sri-csl-95-02/`.
15. Poledna, S., Burns, A., Wellings, A., Barrett, P.: Replica determinism and flexible scheduling in hard real-time dependable systems. IEEE ToC (2) (February 2000)
16. Gammar, S.M., Kamoun, F.: A comparison of scheduling algorithms for real time distributed transactional systems. In: Proc. of 6th IEEE CS Workshop on Future Trends of Distributed Computing Systems, 1997. (October 1997) 257–261
17. Aguilera, M.K., Lann, G.L., Toueg, S.: On the impact of fast failure detectors on real-time fault-tolerant systems. In: DISC, Springer-Verlag (2002) 354–370
18. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. JACM **43**(2) (1996) 225–267
19. Chen, W., Toueg, S., Aguilera, M.K.: On the quality of service of failure detectors. IEEE ToC **51**(5) (May 2002) 561–580
20. Lynch, N.: Distributed Algorithms. Morgan Kaufmann (1996)
21. NetEm: Netem Wiki `http://linux-net.osdl.org/index.php/Netem`.
22. JSR-1 Expert Group: Real-time specification for Java `http://rtsj.org`.
23. Li, P., Ravindran, B., et al.: A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems. IEEE Trans. Software Engineering **30**(9) (Sept. 2004) 613 – 629
24. Mills, D.L.: Improved algorithms for synchronizing computer network clocks. IEEE/ACM TON **3** (June 1995) 245–254